Grouper Integration Example

- Overview
- Tagging
- Attribute Resolver
    - Supporting Beans
    - Data Connector
    - Entitlement Attribute
- Attribute Filter

Overview

This is an outline of a real world example of integration between the IdP and the Grouper software using the HTTPConnector introduced in IdP V3.4. LDAP is commonly used as a go-between for the two systems, but you can skip that step if your organization is better at running Grouper's Web Services than LDAP.

The scenario illustrated in this article is:

- An SP is "tagged" with a metadata attribute indicating that group lookup should be enabled.
- A web service call to Grouper is made to retrieve group memberships for the subject that are within a stem named after the hashed entityID of the SP.
- The groups are transformed into uniquely named eduPersonEntitlement values that prevent collisions between identically-named groups across different services.
- An AttributeFilterScript rule releases entitlements corresponding to the SP.

Once implemented, the scenario automates group lookup and entitlement release for all tagged services (with little or no overhead for any non-tagged services).

Tagging

The metadata extension attribute used in this example is the following:

```
<saml:Attribute Name="http://shibboleth.net/ns/attributes/releaseAllValues"
                NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
        <saml:AttributeValue>grouperGroups</saml:AttributeValue>
</saml:Attribute>
```

This can be embedded directly in an `<mdattr:EntityAttributes>` extension, or added at runtime to an external metadata source using the EntityAttributesFilter feature.

This tag will be used in other parts of the example to control activation of components and filtering.

Attribute Resolver

Most of the complex bits are in the resolver obviously, to pull in Grouper data and format it into entitlements. Most of this is accomplished with scripts of various sorts.

A note of caution: these examples are written using the pre-Java 8 Rhino scripting engine because that's what I've stuck with in my deployment as a personal preference (you'll see the `language` attribute used explicitly to highlight this). Using the examples with Nashorn would require some re-writes, but nothing dramatic.

Supporting Beans

Often, advanced use cases require some native Spring wiring for certain kinds of objects, and this is definitely one of those cases. I load beans needed for the resolver in a separate Spring file added as an additional resource to *conf/services.xml*.

There are a variety of beans here, divided into two categories:

- Some fairly simple beans needed to control or customize the behavior of the HTTPConnector itself.
- More complex beans that define a custom-purpose HttpClient for the web service calls, including specialized security configuration to support pre-emptive HTTP Basic Authentication to authenticate via a service account to Grouper.

The custom client provides for better handling of timeout behavior, and the security beans provide appropriate TLS validation of the server, and avoid extra round trips by providing the HTTP credentials in every call instead of waiting for a 403 challenge from the server.

A description of some of the beans follows the example.

attribute-resolver-spring.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:util="http://www.springframework.org/schema/util"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:c="http://www.springframework.org/schema/c"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans
/spring-beans.xsd
                        http://www.springframework.org/schema/context http://www.springframework.org/schema
/context/spring-context.xsd
                        http://www.springframework.org/schema/util http://www.springframework.org/schema/util
/spring-util.xsd"

    default-init-method="initialize"
    default-destroy-method="destroy">

    <bean id="shibboleth.IdentifiableBeanPostProcessor"
        class="net.shibboleth.ext.spring.config.IdentifiableBeanPostProcessor" />

    <bean id="pathEscaper" class="com.google.common.net.UrlEscapers" factory-method="urlPathSegmentEscaper" />

    <!-- Used as activationCondition-ref in resolver "proper" to trigger lookup with a tag. -->

    <bean id="GrouperCondition" parent="shibboleth.Conditions.EntityDescriptor">
        <constructor-arg name="pred">
            <bean class="org.opensaml.saml.common.profile.logic.EntityAttributesPredicate">
                <constructor-arg>
                    <list>
                        <bean class="org.opensaml.saml.common.profile.logic.EntityAttributesPredicate.Candidate"
                            c:name="http://shibboleth.net/ns/attributes/releaseAllValues"
                            p:values="grouperGroups" />
                    </list>
                </constructor-arg>
            </bean>
        </constructor-arg>
    </bean>

        <!-- HttpClient bean for web service calls. -->

        <bean id="grouperHttpClient" lazy-init="true"
        class="net.shibboleth.idp.profile.spring.relyingparty.metadata.HttpClientFactoryBean"
        p:maxConnectionsPerRoute="20"
        p:maxConnectionsTotal="20"
        p:connectionTimeout="PT2S"
        p:connectionRequestTimeout="PT2S"
        p:socketTimeout="PT5S"
        p:tLSSocketFactory-ref="shibboleth.SecurityEnhancedTLSSocketFactory" />

        <!-- Security parameters for HTTP client. -->

    <bean id="grouperHttpSecurity" lazy-init="true"
            class="org.opensaml.security.httpclient.HttpClientSecurityParameters"
            p:authCache-ref="grouperAuthCache">
        <property name="tLSTrustEngine">
            <bean class="net.shibboleth.idp.profile.spring.factory.StaticPKIXFactoryBean"
                p:checkNames="true"
                p:trustedNames="*.service.osu.edu"
                p:verifyDepth="3"
                p:certificates="%{idp.home}/credentials/usertrust.pem" />
        </property>
    </bean>

        <!-- Injects basic-auth credentials into security parameter bean using a method call. -->

    <bean class="org.springframework.beans.factory.config.MethodInvokingBean"
            p:targetObject-ref="grouperHttpSecurity"
            p:targetMethod="setBasicCredentialsWithScope">
        <property name="arguments">
```

```
            <list>
                <bean class="org.apache.http.auth.UsernamePasswordCredentials" c:_0="webauth" c:_1="%{idp.
grouper-ws.password}" />
                <bean class="org.apache.http.auth.AuthScope" c:_0="grouper-ws.service.osu.edu" c:_1="443" />
            </list>
        </property>
    </bean>

    <!-- Primes the basic-auth cache for the target host. -->

    <bean id="grouperAuthCache" lazy-init="true" class="org.apache.http.impl.client.BasicAuthCache" />

    <bean class="org.springframework.beans.factory.config.MethodInvokingBean"
            p:targetObject-ref="grouperAuthCache"
            p:targetMethod="put">
        <property name="arguments">
            <list>
                <bean class="org.apache.http.HttpHost" c:_0="grouper-ws.service.osu.edu" c:_1="443" c:_2="
https" />
                <bean class="org.apache.http.impl.auth.BasicScheme" />
            </list>
        </property>
    </bean>

        <!-- Custom object used to hash SP entityIDs. -->

    <bean id="osu.StringDigester" class="net.shibboleth.utilities.java.support.codec.StringDigester" c:
algorithm="SHA1" c:format="HEX_LOWER" />

</beans>
```

I'm not going to cover all of this because you should be able to look up the Javadocs for most of it yourself, but the HTTP client security is fairly involved and needs explanation. It's easier to explain from the bottom up, and it's also useful if you're familiar with the "MethodInvokingBean" class. This is a Spring trick that allows a bean to be defined solely for the purpose of calling a Java method on another bean, when it's necessary to do things that aren't designed for the Dependency Injection model.

The bottom of the stack here is a the HttpClient "AuthCache", which is a way of priming the client with information about specific web servers that require authentication so that it doesn't wait to be prompted by the server and cause extra round trips. This is only safe to do if you have good control over the TLS validation process to make sure connections are only made to trusted servers, which is handled later.

The "grouperAuthCache" bean is the cache itself, and we have to call the `put` method on it with a pair of arguments that define the host parameters and the type of authentication to use. The credentials themselves are handled separately, this just tells the client "use basic-auth when accessing this host".

The "grouperHttpSecurity" bean is the Shibboleth-defined object that encapsulates the securing of the HTTP client. Some of the aspects of this class are discussed on the HttpClientConfiguration page. The cache mentioned above is injected into one property. The basic-auth credentials themselves are injected by calling the `setBasicCredentialsWithScope` method, which includes a "scope" that matches the host identified in the cache, tying all of that together.

Finally, a trust engine is injected that handles verification of the server's certificate. The CA certificate is defined with a property, and the depth of the chain is extended to allow for intermediate CAs. Because the built-in code does not understand wildcard certificates (which are a bad thing in general, but...) the name of the certificate has to be manually added to the configuration to allow it to be matched.

Note that the "grouperHttpClient" and "grouperHttpSecurity" beans are not tied together here. In general, client objects are independent of security, and the security features are applied separately to the object using the client (in this case the eventual HTTPConnector). This allows a single client to be used against different servers if appropriate.

Data Connector

The most interesting part of this example is the HTTPConnector itself, and how it forms the request and handles the response. For convenience the script that parses the JSON response is in a separate file, so the connector itself is shorter:

Connector in attribute-resolver.xml

```
    <DataConnector id="grouperGroups" xsi:type="HTTP"
        activationConditionRef="GrouperCondition"
        propagateResolutionExceptions="false"
        httpClientRef="grouperHttpClient"
        httpClientSecurityParametersRef="grouperHttpSecurity"
        acceptTypes="text/x-json">

        <InputAttributeDefinition ref="IDMUID" />

        <URLTemplate customObjectRef="osu.StringDigester">
          <![CDATA[
          https://grouper-ws.service.osu.edu/gms-ws/servicesRest/v2.3.000/subjects/#if($IDMUID && $IDMUID.size()
== 1)$IDMUID.get(0)#end/groups?wsLiteObjectType=WsRestGetGroupsLiteRequest&stemName=OSU%3AWebLoginService%
3A$custom.apply($resolutionContext.getAttributeRecipientID())%3Aapp&stemScope=ALL_IN_SUBTREE
          ]]>
        </URLTemplate>

        <ResponseMapping language="rhino-nonjdk">
          <ScriptFile>%{idp.home}/conf/grouperGroups.js</ScriptFile>
        </ResponseMapping>
    </DataConnector>
```

The main element contains a lot of the references to the objects defined in the other file, mentioned earlier, like the activation condition, the HTTP client bean, and the security bean. The other interesting property is one that causes the client to advertise it supports JSON using a MIME type that Grouper understands. Without that, the web service returns XML instead.

The bottom part just defines the script file to run to process the response and is covered later.

The <URLTemplate> element is the interesting bit but it's mostly obvious except for the encoding of the entityID and if you weren't aware that, as a Velocity template, these kinds of query templates can do conditionals, though it's a bit unreadable. The conditional just error-guards a missing input attribute. The stem for the search here is the part that standardizes the interaction. We hash the entityID using the custom bean defined earlier, so the stem format in Grouper is "OSU:WebLoginService:<hashed>:app". The hashing avoids encoding issues in Grouper, and we use display names on that side to hide them from the application owners.

The response parsing script is below, and it's a "real-time" handler, meaning it consumes the data exactly once, so it's suitable for streaming responses without a content length. The helper method enforces a limit on size as it consumes the data.

grouperGroups.js

```
importClass(Packages.java.util.ArrayList);
importClass(Packages.net.shibboleth.utilities.java.support.httpclient.HttpClientSupport);
importPackage(Packages.net.shibboleth.idp.attribute);

// This limits the body size for security purposes (could be expanded for large group counts).
var body = HttpClientSupport.toString(response.getEntity(), "UTF-8", 65536);

// The JSON object is built-in to JavaScript, which is obviously why everybody wants JSON.
var result = JSON.parse(body);

if (result != null
        && result.WsGetGroupsLiteResult != null
        && result.WsGetGroupsLiteResult.resultMetadata != null
        && result.WsGetGroupsLiteResult.resultMetadata.resultCode == "SUCCESS") {
        var groups = result.WsGetGroupsLiteResult.wsGroups;
        if (groups != null) {
                var values = new ArrayList();
                for (var i=0; i<groups.length; i++) {

                        // Format is OSU:WebLoginService:SHA1(entityID):app:grouppath

                        // If we find any OSU:AWS prefixes, we got back "everything" from a
                        // query on a missing stem. Easy way to check for that case.
                        if (groups[i].name.lastIndexOf("OSU:AWS:", 0) == 0) {
                                values.clear();
                                break;
                        }

                        var segments = groups[i].name.split(":");
                        if (segments.length >= 5) {
                                var name = segments[4];
                                for (var j = 5; j < segments.length; j++) {
                                        name = name + '/' + segments[j];
                                }
                                values.add(new StringAttributeValue(name));
                        }
                }

                if (!values.isEmpty()) {
                        var GROUP = new IdPAttribute("GROUP");
                        GROUP.setValues(values);
                        connectorResults.add(GROUP);
                }
        }
}
```

Most of this is just parsing the Grouper result, but some interesting points:

- Grouper has a weird bug that causes a request for a non-existent stem to return every group membership the requesting account can see. To prevent this, the code checks for one particular stem we happen to have separately defined for Amazon groups, so it can tell from that result that the data should be cleared and nothing returned.
- The convention used is to strip the fixed stem prefix and then convert colons into slashes as a separator, which will translate better into entitlement URLs.

The final result is created as an attribute called "GROUP", containing the reformatted group names.

Entitlement Attribute

This isn't the entire script we use for the eduPersonEntitlement attribute, but the relevant portion is used to translate the values of the "GROUP" attribute from the connector into the URLs passed out to applications.

The "custom" object in this script is the "pathEscaper" bean that's defined up above in my supporting bean set. This encodes group name segments for URL safety.

eduPersonEntitlement.js

```
importClass(Packages.java.lang.StringBuilder);

if (typeof GROUP != "undefined" && GROUP != null && GROUP.getValues() != null) {
    var iter = GROUP.getValues().iterator();
    while (iter.hasNext()) {
        var splitted = iter.next().split("/");
        if (splitted.length > 0) {
            var name = new StringBuilder(resolutionContext.getAttributeRecipientID());
            if (name.charAt(name.length() - 1) != '/')
                name.append('/');
            name.append(custom.escape(splitted[0]));
            for (var i = 1; i < splitted.length; i++) {
                name.append('/').append(custom.escape(splitted[i]));
            }
            eduPersonEntitlement.getValues().add(name.toString());
        }
    }
}
```

The script builds up entitlement values by appending the group names to the SP's entityID as a "root" URL. This doesn't account for URNs, but we don't generally have many SPs named with URNs and I could special-case them if I had to.

A simple example: a Slack users group for an SP named https://ohio-state.slack.com is named "OSU:WebLoginService:0df6144ed33b45dc50c4f0ac402100feb4ec01e2:app:users" in Grouper, and the entitlement becomes https://ohio-state.slack.com/users

This is the basic philosophy I took: nobody's going to avoid conflicting group names like "users", so every group should be inherently namespaced by the SP itself. This doesn't work so well with massively vhosted SPs that all rely on a single entityID, but we'll cross that bridge if it comes up, probably by subdividing the apps stem for different delegated owners in Grouper, and building in naming separation under that layer.

Attribute Filter

The final piece of this "hands off" solution was to come up with a filter policy rule to auto-release exactly the right groups to each SP that was tagged. A script was the simplest way to do that too:

attribute-filter.xml

```
    <AttributeFilterPolicy id="Per-Attribute-grouperGroups">
        <PolicyRequirementRule xsi:type="EntityAttributeExactMatch"
                       attributeName="urn:mace:osu.edu:shibboleth:attribute-release"
                       attributeValue="grouperGroups" />

        <AttributeRule attributeID="eduPersonEntitlement">
            <PermitValueRule xsi:type="Script" language="rhino-nonjdk">
                <Script>
                <![CDATA[
                    importClass(java.util.HashSet);
                    var result = new HashSet();
                    var iter = attribute.getValues().iterator();
                    while (iter.hasNext()) {
                        var val = iter.next();
                        if (val.getValue().startsWith(filterContext.getAttributeRecipientID())) {
                            result.add(val);
                        }
                    }
                    result;
                ]]>
                </Script>
            </PermitValueRule>
        </AttributeRule>
    </AttributeFilterPolicy>
```

The script just releases any value that starts with the SP's name.